CS10720 Problems and Solutions

Thomas Jansen

Today: Quick Sort
Quick Select

March 14th

Plans for Today

Partitioning

Remember

- start looking for incorrectly placed items left and right of the pivot element and swap if found
- need to consider pivot element itself 'incorrectly placed' otherwise it may not reach its correction position
- need to keep track of position of pivot element which can change if it gets swapped
- need to force to move on to avoid being caught in endless loop

Setting work in array `int keys[size]`
have initially `left` denote the leftmost index
have initially `right` denote the rightmost index
have pivot give index of pivot element

Implementing Partitioning

Setting work in `int keys[size]`; `pivot` gives index of pivot element
have initially `left/right` denote leftmost/rightmost index

```
long partition(long *keys, long left, long right, long pivot) {
  long swap; /* used for key swap */
  while ( left < right ) {
    while ( keys[left] < keys[pivot] )
      left++; /* search for wrongly placed item */
    while ( keys[right] > keys[pivot] )
      right--; /* search for wrongly placed item */
    /* swap items */
    swap = keys[left];
    keys[left] = keys[right];
    keys[right] = swap;
    /* check if pivot was swapped */
    if ( left == pivot )
      pivot=right; /* update pivot position */
    else if ( right==pivot )
      pivot=left; /* update pivot position */
    if ( left<pivot )
      left++; /* force index to move if different from pivot */
    if ( right>pivot )
      right--; /* force index to move if different from pivot */
  }
  return pivot; /* return updated pivot position */
}
```

Quick Sort

```
void quickSort(long *keys, long start, long size) {
  long pivot; /* index of pivot element */
  if ( size < 2 )
    return; /* nothing to do for arrays of size < 2 */
  pivot = start+(size/2); /* select some pivot element */
  pivot = partition(keys, start, start+size-1, pivot); /* partition inp
  quickSort(keys, start, pivot-start); /* sort left part */
  quickSort(keys, pivot+1, start+size-pivot-1); /* sort right part */
}
```

Observation

- main work lies in `partition` (plus recursion)
- choice of pivot element completely arbitrary
- recursion depth depends on the size of the different parts

Idea for improvement (to limit recursion depth)
sort the smaller part first

Better Quick Sort

```
void quickSort(long *keys, long start, long size) {
  long pivot; /* index of pivot element */
  if ( size < 2 )
    return; /* nothing to do for arrays of size < 2 */
  pivot = start+(size/2); /* select some pivot element */
  pivot = partition(keys, start, start+size-1, pivot); /* partition input */
  if ( pivot-start < size-pivot-1) { /* check size of parts */
    quickSort(keys, start, pivot-start); /* sort smaller part */
    quickSort(keys, pivot+1, start+size-pivot-1); /* sort larger part */
  }
  else {
    quickSort(keys, pivot+1, start+size-pivot-1); /* sort smaller part */
    quickSort(keys, start, pivot-start); /* sort larger part */
  }
}
```

Observation

- main work lies in `partition` (plus recursion)
- choice of pivot element completely arbitrary
- recursion depth depends on the size of the smaller part
  if the compiler is clever enough to handle second recursion as iteration (know as tail call optimization)

Analysing Quick Sort Part 1: partition

```
long partition(long *keys, long left, long right, long pivot) {
  long swap; /* used for key swap */
  while ( left < right ) {
    while ( keys[left] < keys[pivot] )
      left++; /* search for wrongly placed item */
    while ( keys[right] > keys[pivot] )
      right--; /* search for wrongly placed item */
    /* swap items */
    swap = keys[left]; keys[left] = keys[right]; keys[right] = swap;
    /* check if pivot was swapped */
    if ( left == pivot )
      pivot=right; /* update pivot position */
    else if ( right==pivot )
      pivot=left; /* update pivot position */
    if ( left<pivot )
      left++; /* force index to move if different from pivot */
    if ( right>pivot )
      right--; /* force index to move if different from pivot */
  }
  return pivot; /* return updated pivot position */
}
```

Observations

- `left` never decreased, `right` never increased
- in each round at least either `left` increased or `right` decreased (in many rounds both and by more than only 1)
- stops when $\texttt{left} \geq \texttt{right}$
- run time $\Theta(\texttt{right} - \texttt{left})$

Analysing Quick Sort Part 2: Main Algorithm

```
void quickSort(long *keys, long start, long size) {
  long pivot; /* index of pivot element */
  if ( size < 2 )
    return; /* nothing to do for arrays of size < 2 */
  pivot = start+(size/2); /* select some pivot element */
  pivot = partition(keys, start, start+size-1, pivot); /* partition input */
  if ( pivot-start < size-pivot-1) { /* check size of parts */
    quickSort(keys, start, pivot-start); /* sort smaller part */
    quickSort(keys, pivot+1, start+size-pivot-1); /* sort larger part */
  }
  else {
    quickSort(keys, pivot+1, start+size-pivot-1); /* sort smaller part */
    quickSort(keys, start, pivot-start); /* sort larger part */
  }
}
```

Remember Let $n =$ size, $T(n)$ total run time

Observation $T(n) = \Theta(n) + T(s) + T(n-s-1)$
where s depends on position of pivot after partition

Analysing Quick Sort Part 3: Result

Remember $n = \texttt{size}$, $T(n)$ total run time
$T(n) = \Theta(n) + T(s) + T(n-s-1)$
where s depends on position of pivot after partition

Consider extreme case $s = 1$ always

$$\begin{aligned} T(n) &= \Theta(n) + T(n-2) \\ &= \Theta(n) + \Theta(n-2) + T(n-4) \\ &\dots \\ &= \Theta(n) + \Theta(n-2) + \Theta(n-4) + \cdots + \Theta(1) \\ &= \Theta(n^2) \end{aligned}$$

Fact worst case

Consider extreme case $s = n/2$ always
equal to Merge Sort
$T(n) = \Theta(n \log n)$

Can we improve the worst case?

Observation need to avoid bad luck with the choice of the pivot

Avoiding Bad Luck On Average

Fact one can have bad luck once or twice or even a couple of times
but in the long run things will 'even out'

Idea select pivot element randomly

How is this different from choosing a fixed position?

Observation randomising the selection of the pivot
moves your reliance of 'bad luck avoidance' from input
to the random choices made

Why is this helping?

Observation input can have structure
structure of input may be precisely bad for algorithm
random choices have no structure
so having bad luck often is very unlikely

Randomised Quick Sort

```
void quickSort(long *keys, long start, long size) {
  long pivot; /* index of pivot element */
  if ( size < 2 )
    return; /* nothing to do for arrays of size < 2 */
  pivot = start+(rand()%size); /* select random pivot; needs RAND_MAX>size */
  pivot = partition(keys, start, start+size-1, pivot); /* partition input */
  if ( pivot-start < size-pivot-1) { /* check size of parts */
    quickSort(keys, start, pivot-start); /* sort smaller part */
    quickSort(keys, pivot+1, start+size-pivot-1); /* sort larger part */
  }
  else {
    quickSort(keys, pivot+1, start+size-pivot-1); /* sort smaller part */
    quickSort(keys, start, pivot-start); /* sort larger part */
  }
}
```

Observation algorithmically almost identical
(definitely not more difficult to implement)

Extra Information: Analysing Randomised Quick Sort

Remember $n =$ `size`, $T(n)$ total run time
$T(n) = \Theta(n) + T(s) + T(n-s-1)$
where s depends on position of pivot after partition

Now take average over random choices
because they are random and will average out (over time)

$$
\begin{aligned}
T(n) &= \sum_{s=0}^{n-1} \frac{1}{n} \cdot (\Theta(n) + T(s) + T(n-s-1)) \\
&= \left(\sum_{s=0}^{n-1} \frac{1}{n} \cdot \Theta(n) \right) + \sum_{s=0}^{n-1} \frac{1}{n} \cdot (T(s) + T(n-s-1)) \\
&= \Theta(n) + \sum_{s=0}^{n-1} \frac{1}{n} \cdot (T(s) + T(n-s-1)) \\
&= \Theta(n) + \frac{1}{n} \sum_{s=0}^{n-1} T(s) + T(n-s-1) \overset{(*)}{=} \Theta(n \log n)
\end{aligned}
$$

(if you *really* want to see (*) see next slide)

Observation randomisation improves worst case to $\Theta(n \log n)$
i. e., while Quick Sort is slow in the worst case
randomised Quick Sort is fast in the worst case

Solving Recurrence for Randomised Quick Sort Analysis

Remember $T(n) = \Theta(n) + \frac{1}{n}\sum_{s=0}^{n-1} T(s) + T(n-s-1)$

simplify this a bit to $T(n) = n + \frac{1}{n}\sum_{s=0}^{n-1} T(s) + T(n-s-1)$

We conclude $n \cdot T(n) = n^2 + \sum_{s=0}^{n-1} T(s) + T(n-s-1)$ and

$(n-1) \cdot T(n-1) = (n-1)^2 + \sum_{s=0}^{n-2} T(s) + T(n-s-2)$

Thus $n \cdot T(n) - (n-1) \cdot T(n-1) = 2n + 2T(n-1)$ and
$n \cdot T(n) - (n+1)T(n-1) = 2n$ and also
$\frac{T(n)}{n+1} - \frac{T(n-1)}{n} = \frac{2}{n+1}$ so we have $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1}$

Define $D(n) = \frac{T(n)}{n+1}$ and have (with $D(1) = 1$)
$D(n) = D(n-1) + \frac{2}{n+1} = D(n-2) + \frac{2}{n} + \frac{2}{n+1}$
$= D(n-3) + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} = \cdots = 1 + \sum_{i=1}^{n+1} \frac{2}{i}$
$< 3 + 2\ln(n)$ (since $\sum_{i=1}^{n+1} \frac{1}{i}$ is the $(n+1)^{\text{th}}$ harmonic number)

With $D(n) = \Theta(\log n)$ and $T(n) = (n+1)D(n)$
we have $T(n) = \Theta(n \log n)$ as claimed $\square$

Selection Problem

Input array `long keys[size]` and `int r`
Output position `int p` such that `keys[p]` is r^{th} smallest item in `keys`

Examples $r = 1$ yields minimum, $r =$ size yields maximum
$r =$ size$/2$ yields median, $r =$ size$/4$ yields lower quartile

Observation We already know how to solve this!

1. Sort the array `keys`.
2. Return `keys[r-1]`.

Observation takes time $\Theta(n \log n)$ (with $n =$ `size` as usual)
in the worst case and the average case

Can we do this faster? (Can we do this in time $o(n \log n)$?)

Randomised Quick Sort

```
void quickSort(long *keys, long start, long size) {
  long pivot; /* index of pivot element */
  if ( size < 2 )
    return; /* nothing to do for arrays of size < 2 */
  pivot = start+(rand()%size); /* select random pivot; needs RAND_MAX>size */
  pivot = partition(keys, start, start+size-1, pivot); /* partition input */
  if ( pivot-start < size-pivot-1) { /* check size of parts */
    quickSort(keys, start, pivot-start); /* sort smaller part */
    quickSort(keys, pivot+1, start+size-pivot-1); /* sort larger part */
  }
  else {
    quickSort(keys, pivot+1, start+size-pivot-1); /* sort smaller part */
    quickSort(keys, start, pivot-start); /* sort larger part */
  }
}
```

Remember after `partition` pivot element at correct place

Consequence part in which element with rank r is is known
no need to sort the other part

Idea continuing search in only one part
may be significantly faster than sorting

Randomised Quick Select

```
long quickSelect(long *keys, long start, long size, long r) {
   long pivot; /* index of pivot element */
   if ( size < 2 )
     return start; /* only item in array of size 1 must be it */
   pivot = start+(rand()%size); /* select random pivot; needs RAND_MAX>size */
   pivot = partition(keys, start, start+size-1, pivot); /* partition input */
   if ( pivot+1 == r )
     return pivot; /* found the correct item */
   if ( pivot+1 > r) /* search in part with smaller keys */
     return quickSelect(keys, start, pivot-start, r);
   else /* search in part with greater keys */
     return quickSelect(keys, pivot+1, start+size-pivot-1, r);
   }
}
```

Remark comparison with pivot+1 (instead of pivot)
because in sorted array item with rank 1 sits at position 0

See $T(n) =$ worst case run time of quick select on array of size n
$T(n) = \Theta(n) + T(s)$ (s is size of part in recursion)
because `partition` has run time $\Theta(n)$

Extra Information: Analysis Quick Select

Remember $T(n) = \Theta(n) + T(s)$
(simplified to $T(n) = n + T(s)$ to make our life simpler)

Remember we average over random events because they are random

Consequence $$T(n) = n + \sum_{s=0}^{n-1} \frac{1}{n} \cdot T(s)$$

$\Leftrightarrow$ $$nT(n) = n^2 + \sum_{s=0}^{n-1} T(s)$$

also $$(n-1)T(n-1) = (n-1)^2 + \sum_{s=0}^{n-2} T(s)$$

Cool trick
$$\begin{aligned} & nT(n) - (n-1)T(n-1) \\ &= \left(n^2 + \sum_{s=0}^{n-1} T(s)\right) - \left((n-1)^2 + \sum_{s=0}^{n-2} T(s)\right) \\ &= n^2 - (n-1)^2 + T(n-1) = 2n - 1 + T(n-1) \end{aligned}$$

We now have $$nT(n) - (n-1)T(n-1) = 2n - 1 + T(n-1)$$

Extra Information: Analysis Quick Select (cont.)

Remember $nT(n) - (n-1)T(n-1) = 2n - 1 + T(n-1)$

$\Leftrightarrow$ $nT(n) = nT(n-1) + 2n - 1$

$\Leftrightarrow$

$$
\begin{aligned}
T(n) &= T(n-1) + 2 - 1/n \\
&= T(n-2) + 2 - 1/(n-1) + 2 - 1/n \\
&= T(n-3) + 2 + 1/(n-2) + 2 - 1/(n-1) + 2 - 1/n \\
&\vdots \\
&= T(n-k) + k \cdot 2 - \sum_{i=0}^{k-1} 1/(n-i) \\
&= T(1) + 2(n-1) - \sum_{i=0}^{n-2} 1/(n-i) \\
&= \Theta(n)
\end{aligned}
$$

Remember this! (and include in your portfolio)

Theorem

Quick select can find an element of rank r in an array of size n in the worst case in expected time $\Theta(n)$.

Summary & Take Home Message

Things to remember

- quick sort
- guarding against the worst case: randomisation
- selection problem, quick select and analysis

Take Home Message

- Quick sort is very efficient in the average case and very inefficient in the worst case.
- Randomised quick sort is very efficient in the expected case.
- Solving specific problems can sometimes be done more efficient than solving the general problem.
- Exploiting the obvious is sometimes all the cleverness it takes.

Lecture feedback `http://onlineted.com`